# Monads, Zippers and Views

## Virtualizing the Monad Stack

Tom Schrijvers
Department of Applied Mathematics and Computer Science, Universiteit Gent
tom.schrijvers@ugent.be

Bruno C. d. S. Oliveira
ROSAEC Center, Seoul National University
bruno@ropas.snu.ac.kr

## Abstract

We make monadic components more reusable and robust to changes by employing two new techniques for *virtualizing* the monad stack: the *monad zipper* and *monad views*. The *monad zipper* is a higher-order monad transformer that creates *virtual* monad stacks by ignoring particular layers in a concrete stack. *Monad views* provide a general framework for monad stack virtualization: they take the monad zipper one step further and integrate it with a wide range of other virtualizations. For instance, particular views allow restricted access to monads in the stack. Furthermore, monad views provide components with a *call-by-reference*-like mechanism for accessing particular layers of the monad stack.

With our two new mechanisms, the monadic effects required by components no longer need to be literally reflected in the concrete monad stack. This makes these components more reusable and robust to changes.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

***General Terms*** Design, Languages

***Keywords*** Components with Side Effects, Modularity, Monad Transformers, Zipper

## 1. Introduction

*Monads* [15, 31] are a useful abstraction for encapsulating side-effects in purely functional languages like Haskell [18]. With monads different types of effects – such as state, non-determinism or exceptions – can be modeled with the same abstract interface.

Monads are composed via mechanisms such as monad transformers [13]. With monad transformers writing programs that use multiple effects is possible by stacking different transformers on top of each other to form a larger monad. Monads and monad transformers are interesting because they allow programmers to write realistic effectful programs that are still purely functional, thus enjoying reasoning principles such as *equational reasoning* and *parametricity* [21, 29]. Furthermore, because of these reasoning principles, monads and monad transformers also provide a good setting for studying modular effectful components, like Aspect-Oriented Programming (AOP) style advice [17] or models of Feature-Oriented Programming (FOP) [20].

Monad transformers come with a responsibility: it is necessary to manage the monad stack. In Haskell there are two existing approaches for this: *implicit liftings* and *explicit liftings*. Implicit liftings exploit type-directed overloading of methods, provided by type classes [32], to automatically route an effectful operation to the first layer of the right type in the monad stack. Explicit liftings offer more control to programmers, at the cost of automation, by allowing monadic components to directly refer to a monad layer at some position below the current layer using multiple invocations of the *lift* method (which moves from the current layer to the one immediately below).

However, there are two main problems with the current mechanisms for manipulating the monad stack:

1. **Inconvenience:** The first problem is that explicit lifting is simply "awkward" to use. Explicit lifting relies on a positional mechanism, similar to de Bruijn indices, to refer to particular layers of the monad stack. Because access to the *n*th layer below the current layer requires *n* calls to the *lift* method, code quickly becomes polluted with *lift* method invocations.
2. **Lack of robustness, limited expressiveness and reusability:** The second, and more fundamental, problem is that the current approach is not robust to changes and offers only limited expressiveness and reusability, specially when *higher-order components* (or *control flow* operators) are involved. Implicit lifting through overloaded methods is robust, but it is also quite limited in expressiveness since handling multiple monads of the same type is not possible. Explicit liftings are not robust to changes because the relative (positional) references impose a tight coupling between monadic components and the monad stack, which makes those components less adaptable. Furthermore, explicit lifting still has limited expressiveness because it only allows to refer to layers below the current layer, but not layers above.

The first problem is well-known in the Haskell community and there have been some proposals for solving it. Piponi [19] and Snyder & Alexander [25] suggested a solution by labeling monad transformers with (type-level) tags, allowing particular layers in a monad stack to be accessed by name rather than by position, eliminating the pollution arising from multiple calls to *lift*.

The second problem is a bit more subtle because it only shows up when monadic components are meant to be reused in different contexts with different monad stack layouts as, for example, in the modular effectful components studied by Prehofer [20] and Oliveira et al. [17]. For this to be possible, monadic components should abstract over the monad stack by keeping the type represent-



*ICFP'11,* September 19–21, 2011, Tokyo, Japan.

ing the monad stack polymorphic (though constrained). The use of names is also useful to solve this second problem because names are more robust to changes in the monad layout. However, existing tagged approaches [19, 25] are invasive, in the sense that monadic components have to be written specially with tagging in mind, and they still have reusability limitations (such as name clashes).

***Contributions*** This paper proposes two new techniques for manipulating monad stacks: the *monad zipper* and *monad views*. The *monad zipper* is a monad transformer that allows ignoring particular layers in the concrete stack. *Monad views* abstract the concrete monad stack into a virtual monad stack, which presents itself with a more suitable interface to a particular component. Views also allow restricting access (in the sense of permissions) to particular layers of the monad. An important characteristic of both techniques is that they can be used non-invasively, by being applied externally and effectively providing the component with a tailored virtual monad stack. When there are conflicts that are local to a certain component, monad views can also be used invasively to provide a *call-by-reference* mechanism to refer to particular layers in the monad stack. By working with virtual stacks, requirements of components in terms of the monad stack shape no longer need to be literally reflected in the concrete monad stack, making these components more reusable and robust to changes.

We also briefly report on a non-trivial application of our techniques, developed by Schrijvers et al. [24], which shows that the techniques proposed by us scale well in terms of the number of monads in a monad stack. In that application 30 monadic components and equally as many monad transformers were used. We are not aware of any Haskell projects using as many transformers at the same time.

## 2. Overview

The presentation of our work will be made in Haskell, and we will assume knowledge of the language. Nevertheless, as discussed in more detail in Section 7.1, we believe that none of the main concepts presented by us (the monad zipper, views and masks) are Haskell-specific.

In this section we briefly introduce the Haskell monad transformer library (MTL), provide an overview of the current state-of-the-art in manipulating monad stacks and illustrate how the monad zipper and monad views improve upon that state-of-the-art by virtualizing the monad stack.

### 2.1 Quick Monad Transformers Reference

Our ideas and examples are formulated in terms of a variant of the MTL.[1] See Liang et al. [13] for a more in-depth introduction to MTL. Figure 1 summarizes the monad transformers, classes and operations that we use in the course of this paper. The transformers consist of pure computations ($\mathbb{I}_\mathrm{T}$, the identity monad transformer) and computations with a read-only environment ($\mathbb{R}_\mathrm{T}$), updateable state ($\mathbb{S}_\mathrm{T}$) and exceptions ($\mathbb{E}_\mathrm{T}$). These transformers are combined into different monad stacks with the identity monad ($\mathbb{I}$) at the bottom. The type classes (denoted with subscript M) constrain a monad stack to provide support for a particular effect, without assuming a particular stack configuration. Each class offers a number of primitive operations, such as *ask* to access the environment for $\mathbb{R}_\mathrm{M}$.

### 2.2 Monadic Components

An important distinction throughout this paper is between monadic *component* and *client* code. The main goal of this paper is to improve the robustness, reusability and convenience of use of the former. At first sight, it will seem that the client code pays the price for this improvement, because it is forced to make the necessary configuration choices for the more adaptable components. Fortunately, the complexity can be neatly hidden behind combinators.

***Component code*** Component code is intended to be reused in different contexts. Typically such code resides in libraries, which are used by clients in various different applications. As two simple examples of monadic components, consider an incrementer and an assertion checker:

```
inc :: 𝕊M Int m ⇒ m Int
inc = do x ← get
         put (x + 1)
         return (x + 1)

assert :: 𝔼M String m ⇒ Bool → m ()
assert test = if test
                then return ()
                else throwError "Assertion failed"
```

What is interesting about these components is that they can be adapted to work with different concrete monad stacks. This is possible because we use parametric polymorphism to abstract over the monad, imposing only the required restrictions on the layout by using (type-class) constraints. For example, what the signature of *inc* tells us is that it can be used by any client whose monad stack supports state.

Components can, of course, be combined into larger components. For example:

```
comp :: (𝕊M Int m, 𝔼M String m) ⇒ m ()
comp = do x ← inc
          assert (x > 0)
```

**Client code** Client code instantiates the monad stack, and is defined by the end-user when using the components to build particular applications. Different stack configurations may be possible:

```
type M1 = 𝕊T Int (𝔼T String 𝕀)
type M2 = 𝔼T String (𝕊T Int 𝕀)
```

In general, different monad stack layouts have different semantics. Consider the following *run* functions for $M_1$ and $M_2$:

```
runM1 :: Int → M1 a → Either String (a, Int)
runM1 n = run𝕀 ∘ run𝔼T ∘ flip run𝕊T n

runM2 :: Int → M2 a → (Either String a, Int)
runM2 n = run𝕀 ∘ flip run𝕊T n ∘ run𝔼T
```

When we use these functions to instantiate the monad in *comp* we can observe the semantic differences between the two monad stacks:

```
> runM1 (−1) comp
Left "Assertion failed"

> runM2 (−1) comp
(Left "Assertion failed", 0)
```

In the case of $M_1$ changes to the state are lost upon throwing an error, while this is not the case for $M_2$.

### 2.3 State-of-the-art Manipulation of the Monad Stack

***Implicit lifting*** The monadic components presented in Section 2.2 use implicit lifting for accessing the right layer in the monad stack. Effectful operations like *get*, *put* or *throwError* are automatically routed to the first layer of the right type in the monad stack, by exploiting the type-based overloading mechanism of type classes. This approach is robust, because the routing automatically adapts to multiple layouts of the monad stack (such as $M_1$ or $M_2$).

[1] Section 7.2 discusses the differences.

```
-- identity transformer
newtype IT m a
IT      :: m a → IT m a
runIT :: IT m a → m a

-- reader transformer
newtype RT e m a
RT      :: (e → m a) → RT e m a
runRT :: RT e m a → e → m a

-- state transformer
newtype ST s m a
ST      :: (s → m (a, s)) → ST s m a
runST :: ST s m a → s → m (a, s)

-- exception transformer
newtype ET x m a
ET      :: m (Either x a) → ET x m a
runET :: ET x m a → m (Either x a)
```

```
-- identity monad
newtype I a
I      :: a → I a
runI :: I a → a

-- reader class
class Monad m ⇒ RM e m | m → e
ask          :: RM e m ⇒ m e

-- state class
class Monad m ⇒ SM s m | m → s
get          :: SM s m ⇒ m s
put          :: SM s m ⇒ s → m ()

-- exception class
class Monad m ⇒ EM x m | m → x
throwError :: EM x m ⇒ x → m a
catchError :: EM x m ⇒ m a → (x → m a) → m a
```

**Figure 1.** MTL quick reference.

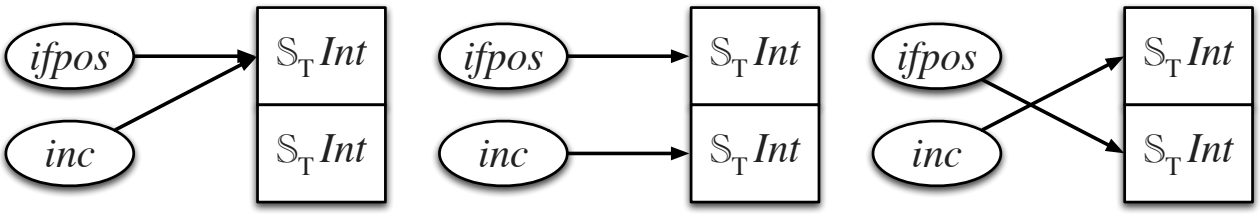


**Figure 2.** Three composition scenarios: *ifpos inc*, *ifpos* (*lift inc*) and ↑ (*ifpos* (↓ *inc*)).

Implicit lifting has one big limitation: it is ineffective for combining multiple instances of the same effect because the automatic type-based selection of a monad layer always picks the first layer of the right type. If there is another layer of that type below that layer, implicit lifting cannot access it.

***Explicit Lifting*** Explicit lifting addresses the limitation of implicit lifting to some extent because it allows moving down the monad stack into a lower layer. Therefore, by calling the effectful operations in lower layers, implicit liftings are routed to monad transformers that are not at the top of the stack.

The approach consists of using *lift* methods *explicitly* to disambiguate the targets of accesses to the monad stack. Suppose that we want to combine two instances of *inc* in such a way that they update different counters. Using explicit lifting we could write a component *doubleInc* that does the job:

```
doubleInc :: (SM Int (t m), SM Int m, MonadTrans t) ⇒ t m ()
doubleInc = inc ≫ lift inc ≫ return ()
```

The role of *lift* is to ensure that the second instance of *inc* updates a counter in the monad below the current layer (and not at the topmost layer). The use of *lift* is reflected in the constraints imposed on the stack. In this case, the requirement is that the top-level monad *t m* must support state and the monad *m* below must also support state. If the monad stack is ST *Int* (ST *Int* I), then *doubleInc* updates the two state layers. Hence, running *doubleInc* with the *run* function below yields the result (((), 1), 6).

```
run c = runI $ runST (runST c 0) 5
```

Explicit lifting is essentially, like de Bruijn indices, a *relative* reference mechanism: it allows moving *n* layers *below*[2] the current layer by using *n* calls to *lift*.

A clear limitation of this mechanism is that it is not possible to refer to layers *above* the current layer. Yet this functionality is particularly desirable to preserve modularity when *higher-order* components (also known as *control flow operators*) are involved. Some compositions of primitive monadic control flow operators (e.g. *catchError*) cannot be expressed with *lift* at all. Furthermore, components that use *lift* internally usually impose unnecessary restrictions on the layout of the monad stack, because they express relative orderings of layers.

In summary implicit liftings are quite robust to changes and are convenient to use, but have limited expressiveness; and, while explicit liftings address the limitations of implicit liftings to some extent, they too are still limited in expressiveness and there is a price to pay, in terms of a tighter coupling between component and monad stack.

[2] Note that there are two points of view for *lift m* in a certain calling context. From the calling context's point of view towards *m* the direction of movement is *downwards*, while, vice versa, the direction of movement from *m*'s point of view towards the calling context is *upwards*. Usually, we adhere to the calling context's point of view, but occasionally we may have to use *m*'s perspective; when that is the case, it should be clear from the context.

### 2.4 Virtualizing the Monad Stack

By presenting the component code with a virtual monad stack, components become decoupled from the client code's concrete monad stack. The monad stack virtualization is achieved by:

1. avoiding *lift* method invocations inside component code, which are responsible for the tight coupling; and
2. using the monad zipper and monad views to manage virtual monad stacks.

In this section we explain how the monad zipper and monad views are used to develop components free of *lift* invocations while, at the same time, allowing for additional expressiveness that is not possible with explicit lifting alone.

Note that, in the remainder of this section, the intention is just to demonstrate how the monad zipper, structural masks, nominal masks and views can be used from the user's point of view to solve various problems related to the manipulation of monad transformers. The reader is not expected to understand in detail how the examples work by the end of Section 2. Instead, the details of each mechanism will be explained in later sections: Section 3 describes how the monad zipper works, Section 4 presents views, and Section 5 presents the different flavors of masks.

***The Monad Zipper*** With the monad zipper we derive a virtual stack from a concrete one, that ignores a prefix but does not forget it. For instance, $M_3$ is a virtual stack for $\mathbb{S}_\mathrm{T}$ *Int* ($\mathbb{S}_\mathrm{T}$ *Int* $\mathbb{I}$) that ignores, but does not forget, the topmost transformer:

```
type M3 = (ST Int ▷ ST Int) I
```

Ignoring a prefix enables an operation ↑ much like *lift*, while still enabling an inverse ↓ that has no counterpart in the explicit lifting approach. This is useful for adapting higher-order monadic components such as *ifpos*

```
ifpos :: SM Int m ⇒ m Int → m Int
ifpos c = do x ← get
             if x > 0
               then c
               else return 0
```

(which routes the flow of control to *c* when a state is positive) to monad layouts where the component *c* is supposed to access a monad *above* the current layer. With the monad zipper this problem is solved as follows:

```
> run (↑ (ifpos (↓ inc)))
((1, 1), 5)
```

Without the monad zipper, the only way to achieve the same result is to invasively (non-modularly) modify *ifpos* such that the *get* method is lifted: explicit lifting alone is not enough to handle the scenario on the right of Figure 2. The zipper, on the other hand, allows *ifpos* to be reused without any invasive changes to the original code.

***Structural Masks*** Masks take the monad zipper one step further and allow selectively ignoring stack layers at multiple arbitrary positions in the monad stack (not just prefixes). Consider for instance the monad stack

```
type M = ST Int (ST Int (ET String (ET String I)))
```

to which the two components *ifpos* and *comp* ≻ *get* have interleaved access in the composition

```
client :: M Int
client = from m1 (ifpos (to m1 (from m2 (comp ≻ get))))
  where m1 = ■
        m2 = □ ::: ■ ::: ■ ::: □
```

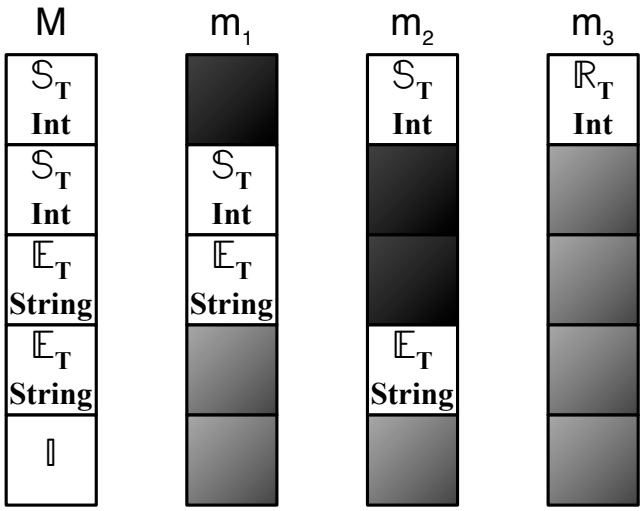


**Figure 3.** The monad stack *M* and the layers visible through masks $m_1$, $m_2$ and $m_3$: black layers are masked and gray layers are abstracted over.

Here, the mask $m_2$ = □ ::: ■ ::: ■ ::: □ has a meaning similar to the bitvector 1001. When imposed on a monad stack, it ignores the second and third layers. The *from m* and *to m* functions, with *m* a mask, generalize ↑ and ↓. For instance, *from* $m_2$ *comp* gives *comp* access to the first and fourth layer in a monad stack. Similarly, $m_1$ provides *ifpos* with access to the second and third layers. See Figure 3 for a graphical depiction. Note that the layers in gray are not masked for the corresponding component, but nor is that component aware of their presence. This is either because a visible layer higher up in the monad stack blocks a similar layer lower down from view, or because the component's polymorphic type does not mention (i.e., abstracts away) the effect of a layer.

***Masks with Restricted Views*** Masks are special instances of a more general mechanism called *views*. As such, masks can be freely combined with other types of views. For instance, the following example shows how to use masks in combination with a read-only view $r_1$, which provides a read-only view of a state monad.

```
client′ :: M Int
client′ = from m1 (ifpos (to m1 (from m2 comp ≻ from m3 ask)))
  where m1 = ■
        m2 = □ ::: ■ ::: ■ ::: □
        m3 = r1
```

The *ask* component, whose type is $\mathbb{R}_\mathrm{M}$ *e m* ⇒ *m e* (see also Figure 1), requires a reader monad, but the concrete monad *M* has no layer with a monad of that kind. However, we can *view* one of the state monads as a reader. The read-only view $r_1$, used in the mask $m_3$, does precisely this and allows the component *ask* to view the first state monad in *M* as a reader monad.

***Nominal Masks*** Referring to layers in a structural fashion (with the monad zipper or bitvector-like masks) can be fragile when the layout of the monad stack is likely to change. With a nominal mask, the client code specifies the names (not the locations) of the layers to be used by a component. That component is then automatically matched up with the correspondingly named layers in the monad stack. For instance, with nominal masks the *client* example would be rewritten as:

```
client″ = ifpos ((comp ≻ get) `use` (Counter1 & Err2))
```

Here, $Counter_1$ and $Err_2$ are the names of respectively the first and fourth stack layer. The combinator *use* takes a list of names and makes the layers tagged with those names visible to the components. The list of names is assembled using the combinator $\&$. The nominal masking infrastructure is built on top of structural masks. The big benefit of using the nominal approach instead is that it is much more robust: if the layers are rearranged, the nominal masks do not have to change at all.

***Call-by-reference with Views*** A final application of views and masks is to allow for a call-by-reference programming style (that is discussed in Section 5.3) in which the view arguments act as references to particular layers in the monad stack.

$$\begin{aligned}&add :: (Monad\ m, \mathbb{S}_{\mathrm{M}}\ Int\ n_1, \mathbb{S}_{\mathrm{M}}\ Int\ n_2)\\&\quad \Rightarrow (n_1 \bowtie m) \to (n_2 \bowtie m) \to m\ ()\\&add\ xref\ yref = \mathbf{do}\ x \leftarrow get_v\ xref\\&\qquad\qquad\quad\ \ y \leftarrow get_v\ yref\\&\qquad\qquad\quad\ \ put_v\ xref\ (x + y)\end{aligned}$$

In this example *xref* and *yref* are two view arguments and the $get_v$ and $put_v$ operations, which generalize the state monad *get* and *put* operations, use such views to access the right layer in the monad stack. This functionality is especially useful when a component employs multiple instances uses of the same effect. By using this call-by-reference style it is possible to avoid the pollution and the ordering constraints on the stack imposed by *lift* methods.

## 3. The Monad Zipper

This section presents the monad zipper: a monad transformer that is used to shift the focus of automatic lifting to the desired layer in the monad stack. In essence, the monad zipper allows us to ignore layers at the top of the stack while, at the same time, preserving these same layers, which allows shifting the focus back to the top when needed.

### 3.1 Stacks and Zippers

Sometimes type-level problems get easier when we move them to the term level. Let's reify the structure of the monad stack in a data type

$$\begin{aligned}&\mathbf{data}\ Stack\ \ = Trans \cdot Stack \mid Bottom\ Monad\\&\mathbf{data}\ Trans\ \ = T_1 \mid \ldots \mid T_n\\&\mathbf{data}\ Monad = I\end{aligned}$$

where the $T_i$ represent the different transformers and $I$ represents $\mathbb{I}$.

Huet [9] taught us how to shift the focus to any position in a data structure, with his *zipper*. Here is the *Zipper* for *Stack*:

$$\begin{aligned}&\mathbf{data}\ Zipper = Zipper\ Path\ Trans\ Stack\\&\mathbf{data}\ Path\ \ \ = Path \blacktriangleright Trans \mid Top\end{aligned}$$

where *Zipper p l s* denotes a stack with layer *l* in focus, remainder of the stack *s* and path *p* back to the top of the stack. The path is a reversed list, where the first element is closest to the layer in focus and the last element is the top of the stack. With a little syntactic sugar

$$(\triangleright) = Zipper$$

we obtain the self-explanatory notation $(Top \blacktriangleright T_1 \blacktriangleright T_2 \blacktriangleright T_3 \triangleright T_4)\ (T_5 \cdot I)$, where the triangles point towards the layer in focus $T_4$.

The *zipper* function turns a stack into a zipper with the first element in focus:

$$\begin{aligned}&zipper :: Stack \to Zipper\\&zipper\ (t \cdot s) = (Top \triangleright t)\ s\end{aligned}$$

while the *up* and *down* functions allow shifting the focus one position up or down:

$$\begin{aligned}&up, down :: Zipper \to Zipper\\&up\ \ \ \ (Zipper\ (p \blacktriangleright t_1)\ t_2\ s) = (p \triangleright t_1)\ (t_2 \cdot s)\\&down\ (Zipper\ p\ t_1\ (t_2 \cdot s)) \ = (p \blacktriangleright t_1 \triangleright t_2)\ s\end{aligned}$$

It's all well and good to zip around a reified form of the monad stack, but can we do it on the real thing too?

### 3.2 Monad Zipper

The answer is yes. Here is how the monad zipper $(\triangleright)$ is defined:

$$\mathbf{newtype}\ (t_1 \triangleright t_2)\ m\ a = \mathbb{Z}_{\mathrm{T}}\ \{run\mathbb{Z}_{\mathrm{T}} :: t_1\ (t_2\ m)\ a\}$$

where the type $(p \triangleright t)\ s$ has similar meaning to the reified data structure above. However, the monad zipper only changes the type representation: the **newtype** indicates that no actual structural change to the monad stack $t_1\ (t_2\ m)$ takes place. The only change takes place in the form of the type, which will enable us to select a different type class instance depending on the layer in focus. We will see more on this later, but first let us complete the analogy between types and terms.

Term-level stack composition $(\cdot)$, as in $T_1 \cdot M_1$, corresponds to type application, as in $t_1\ m_1$. As to $\blacktriangleright$, the type system will not allow terms of type *Zipper* to be used when terms of type *Stack* are expected. This segregation is not the case at the type level: the monad zipper type $(\triangleright)$ can appear as part of a monad stack. Indeed, we define $t_1 \triangleright t_2$ to be the *monad transformer composition* of $t_1$ and $t_2$:

$$\begin{aligned}&\mathbf{instance}\ (MonadTrans\ t_1, MonadTrans\ t_2)\\&\quad \Rightarrow MonadTrans\ (t_1 \triangleright t_2)\ \mathbf{where}\\&\qquad lift = \mathbb{Z}_{\mathrm{T}} \circ lift \circ lift\end{aligned}$$

Hence, at the type level, we simply use $\triangleright$ where $\blacktriangleright$ was needed at the term level. So, the monad stack representation $(t_1 \triangleright \cdots \triangleright t_i)\ (t_{i+1} \cdots m)$ denotes a monad stack with focus on $t_i$.

Finally, analogous to what the *zipper* function does with *Stack*, a monad transformer stack can be transformed into explicit zipper form by the following function:

$$\begin{aligned}&zipper :: t\ m\ a \to (\mathbb{I}_{\mathrm{T}} \triangleright t)\ m\ a\\&zipper = \mathbb{Z}_{\mathrm{T}} \circ \mathbb{I}_{\mathrm{T}}\end{aligned}$$

where the identity monad transformer $\mathbb{I}_{\mathrm{T}}$ acts as the *Top* sentinel.

However, the $\mathbb{I}_{\mathrm{T}}$ sentinel is unnecessary, as the unadorned monad stack $t\ m\ a$ already expresses that the focus rests on $t$. There is no point in adding $\mathbb{I}_{\mathrm{T}}$ to subsequently ignore it again with $\mathbb{I}_{\mathrm{T}} \triangleright t$. In general, $t_1\ (t_2 \cdots (tn\ m))$ represents a monad stack with focus on $t_1$. So we will not actually use the above *zipper* function.

In summary, the term $(Top \blacktriangleright T_1 \blacktriangleright T_2 \blacktriangleright T_3 \triangleright T_4)\ (T_5 \cdot I)$ is the reified form of the type $(t_1 \triangleright t_2 \triangleright t_3 \triangleright t_4)\ (t_5\ \mathbb{I})$.

***Relative Navigation*** Suppose we have a monad transformer stack $t_1\ (t_2 \cdots (t_n\ m))$. Then the focus lies by default on the top-most transformer $t_1$. The monad zipper becomes useful only when we shift the focus away from $t_1$ to $t_2$. The constructor $\mathbb{Z}_{\mathrm{T}}$ accomplishes that shift of focus, but how can we navigate further down, and back up?

Let us start with moving the focus one step further down:

$$\begin{aligned}&step2to3 :: (t_1 \triangleright t_2)\ (t_3\ m)\ a \to (t_1 \triangleright t_2 \triangleright t_3)\ m\ a\\&step2to3 = \mathbb{Z}_{\mathrm{T}}\end{aligned}$$

A further step down:

$$\begin{aligned}&step3to4 :: (t_1 \triangleright t_2 \triangleright t_3)\ (t_4\ m)\ a \to (t_1 \triangleright t_2 \triangleright t_3 \triangleright t_4)\ m\ a\\&step3to4 = \mathbb{Z}_{\mathrm{T}}\end{aligned}$$

The pattern should now be obvious. A single step down at any position in the stack is defined as:

$$\begin{aligned}&\downarrow :: t_1\ (t_2\ m)\ a \to (t_1 \triangleright t_2)\ m\ a\\&\downarrow = \mathbb{Z}_{\mathrm{T}}\end{aligned}$$

Stepping back up is similar:

$$\uparrow :: (t_1 \rhd t_2)\ m\ a \to t_1\ (t_2\ m)\ a$$
$$\uparrow = run\mathbb{Z}_{\mathrm{T}}$$

Finally, note that $\downarrow \circ \uparrow \equiv id$ and $\uparrow \circ \downarrow \equiv id$ hold.

***Focused Behavior*** So far, all we have seen is notation. The interesting *behavior* of $t_1 \rhd t_2$, where it should deviate from a plain monad transformer composition, lies in the methods of the monad classes, e.g. *put* of $\mathbb{S}_{\mathrm{M}}$. For looking up the method implementations it should ignore (look through) $t_1$ and only consider $t_2\ m$. This is achieved, e.g. for the state monad $\mathbb{S}_{\mathrm{M}}$ defined in Figure 1, by lifting the operations through $t_1$:

```
instance (MonadTrans t1, MonadTrans t2, Monad m, SM s (t2 m))
  ⇒ SM s ((t1 ▷ t2) m) where
    get = ZT $ lift $ get
    put = ZT ∘ lift ∘ put
```

Contrast this with GHC's **newtype deriving** construct, that would adopt the same behavior for $(t_1 \rhd t_2)\ m$ as for $t_1\ (t_2\ m)$, e.g., defining the former's *get* in terms of the latter's as $\mathbb{Z}_{\mathrm{T}}\ get$.

In order to generally characterize the required behavior of monad subclass instances such as the above one for $\mathbb{S}_{\mathrm{M}}$, we impose the following law.

Law 1 (Lift Compatibility). *Given any monad subclass constraint $\mathbb{C}_M$ and any computation in this monad subclass $x :: \forall\ m.\mathbb{C}_M\ m \Rightarrow m\ A$, for any type A. Then we must have that*

$$\uparrow (x :: (T_1 \rhd T_2)\ M\ A) \equiv lift\ (x :: T_2\ M\ A)$$

*for any monad transformers $T_1$ and $T_2$, and monad M such that $\mathbb{C}_M\ (T_2\ M)$ holds.*

Observe that this law holds for the above $\mathbb{S}_{\mathrm{M}}$ instance.

As we have no space to provide the details of all monad transformer instances for the monad zipper, we refer the interested reader to our implementations using our variant of the MTL library[3] and the Monatron library [10].[4]

In summary, like explicit lifting, the monad zipper allows us to shift the focus to layers below the current focused layer via $\uparrow$. However, unlike explicit lifting, we can also shift the focus to layers above the focused layer with $\downarrow$. As we saw in Section 2.4 the extra expressiveness of the monad zipper allows for applications which are not possible with explicit liftings.

## 4. Views

The general problem we face when composing effectful components is that of incompatible assumptions about the monad stack. The solution is to work with one concrete monad stack, but to present each component with a suitable virtual monad stack. The correspondence between the concrete and virtual monad stack is captured in a view.[5] In Section 5 we will see how the views framework benefits from the monad zipper.

### 4.1 Virtual Views as Monad Morphisms

Semantically, a view corresponds to the categorical notion of a *monad morphism* (also called *monad transformation* in category theory, not to be confused with Haskell monad transformers that are a special case). Because we will be using different representations for monad morphisms, we capture the essential features in the *MonadCategory* type class.

[3] `http://users.ugent.be/~tschrijv/Haskell/MTLzipper.tgz`

[4] `http://hackage.haskell.org/package/Monatron`

[5] Not to be confused with Wadler's notion of *view* [28].

```
class MonadCategory (⇝) where
  idM  :: (Monad m) ⇒ m ⇝ m
  (•)  :: (Monad l, Monad m, Monad n)
       ⇒ (m ⇝ n) → (l ⇝ m) → (l ⇝ n)
  hmap :: (Monad m, Monad n, MonadTrans t)
       ⇒ (m ⇝ n) → (t m ⇝ t n)
  from :: (Monad m, Monad n) ⇒ (n ⇝ m) → n a → m a
```

The *MonadCategory* class describes a category with monads as objects and monad morphisms as arrows. The identity and composition of the category are $id_M$ and $(\bullet)$, which satisfy the right and left identity, and associativity laws:

$$v \bullet id_M \equiv v$$
$$id_M \bullet v \equiv v$$
$$v_1 \bullet (v_2 \bullet v_3) \equiv (v_1 \bullet v_2) \bullet v_3$$

A Haskell monad transformer $t$ corresponds to an infinite number of monad morphisms in a particular representation $\rightsquigarrow$:

$$\{from^{-1} lift :: m \rightsquigarrow t\ m \mid m \in Monad\}$$

At the same time not all monad morphisms can be expressed by means of Haskell monad transformers. For instance, $\mathbb{R}_{\mathrm{T}}\ \mathbb{I} \rightsquigarrow \mathbb{S}_{\mathrm{T}}\ \mathbb{I}$ does not reflect the application of a monad transformer to a monad.

What characterizes Haskell monad transformers is that they are *functors* over monad morphisms, and *hmap* allows mapping a morphism through a monad transformer, which satisfies the functor laws:

$$hmap\ id_M \equiv id_M$$
$$hmap\ (v_2 \bullet v_1) \equiv hmap\ v_2 \bullet hmap\ v_1$$

The *from* function applies a monad morphism to a monadic computation. A monad morphism preserves the monad structure.

$$from\ v \circ return \equiv return$$
$$from\ v\ (x \gg\!= f) \equiv from\ v\ x \gg\!= from\ v \circ f$$

### 4.2 Uni-directional Views

Uni-directional views constitute the obvious implementation of *MonadCategory*.

```
newtype n ⋉ m = Uni (∀a.n a → m a)
instance MonadCategory (⋉) where
  idM       = Uni id
  v2 • v1   = Uni $ from v2 ∘ from v1
  hmap v    = Uni $ tmap (from v)
  from (Uni v) = v
```

The above implementation is mostly straightforward, but we require a new operation *tmap* supported by all monad transformers to implement *hmap*. To avoid interrupting the flow, we continue with uni-directional views and refer to Section 7.2 for details on *tmap*.

The *lift* function is the most prominent example of a uni-directional view, which turns a monad transformer into the uni-directional view presentation:

```
liftv :: (MonadTrans t, Monad m) ⇒ m ⋉ t m
liftv = Uni lift
```

**Uni-directional views for restricted access** Uni-directional views are useful to restrict access to monadic layers that are shared by multiple components. For example, suppose we wanted to provide particular components only with read access to a shared state. The view $r_1$ can be used to achieve this.

$$r_1 :: (MonadTrans\ t, Monad\ m, \mathbb{S}_M\ s\ (t\ m)) \Rightarrow \mathbb{R}_T\ s\ m \ltimes t\ m$$
$$r_1 = Uni\ \$\ \lambda n \rightarrow \mathbf{do}\ s \leftarrow get$$
$$lift\ \$\ run\mathbb{R}_T\ n\ s$$

In this case the monad transformer $t$ can be any state monad transformer (in particular it can be $\mathbb{S}_T\ s$).

### 4.3 Bi-Directional Views

Some views are invertible. An invertible view is a monad isomorphism, or bi-directional view. We capture bi-directional views in a separate datatype.

$$\mathbf{data}\ n \bowtie m = Bi\ \{from_\bowtie :: \forall a.n\ a \rightarrow m\ a$$
$$, to_\bowtie :: \forall a.m\ a \rightarrow n\ a\}$$

A bi-directional view is of course an instance of *MonadCategory*.

$$\mathbf{instance}\ MonadCategory\ (\bowtie)\ \mathbf{where}$$
$$id_M = Bi\ \{from_\bowtie = id$$
$$, to_\bowtie = id\}$$
$$v_2 \bullet v_1 = Bi\ \{from_\bowtie = from_\bowtie\ v_2 \circ from_\bowtie\ v_1$$
$$, to_\bowtie = to_\bowtie\ v_1 \circ to_\bowtie\ v_2\}$$
$$hmap\ v = Bi\ \{from_\bowtie = tmap\ (from\ v)$$
$$, to_\bowtie = tmap\ (to\ v)\}$$
$$from\ v = from_\bowtie\ v$$
$$to :: (Monad\ n, Monad\ m) \Rightarrow n \bowtie m \rightarrow m\ a \rightarrow n\ a$$
$$to = to_\bowtie$$
$$inverse :: (Monad\ n, Monad\ m) \Rightarrow n \bowtie m \rightarrow m \bowtie n$$
$$inverse\ (Bi\ from\ to) = Bi\ to\ from$$

where

$$v \bullet inverse\ v \equiv id_M \equiv inverse\ v \bullet v$$

**Abstract view constructor** To abstract from the *Uni* and *Bi* representations of bi-directional views, we provide an overloaded *view* constructor function.

$$\mathbf{class}\ MonadCategory\ (\rightsquigarrow) \Rightarrow View\ (\rightsquigarrow)\ \mathbf{where}$$
$$view :: (\forall a.n\ a \rightarrow m\ a) \rightarrow (\forall a.m\ a \rightarrow n\ a) \rightarrow n \rightsquigarrow m$$
$$\mathbf{instance}\ View\ (\ltimes)\ \mathbf{where}\ view\ f\ f^{-1} = Uni\ f$$
$$\mathbf{instance}\ View\ (\bowtie)\ \mathbf{where}\ view\ f\ f^{-1} = Bi\ f\ f^{-1}$$

Using this constructor has the advantage that we can build overloaded views that work not only as bi-directional views, but also as uni-directional views. This is useful to capture isomorphic views like:

$$stateIso :: (Monad\ m, View\ (\rightsquigarrow))$$
$$\Rightarrow (s_2 \rightarrow s_1) \rightarrow (s_1 \rightarrow s_2) \rightarrow \mathbb{S}_T\ s_2\ m \rightsquigarrow \mathbb{S}_T\ s_1\ m$$
$$stateIso\ f\ f^{-1} = view\ (iso\ f\ f^{-1})\ (iso\ f^{-1}\ f)\ \mathbf{where}$$
$$iso\ g\ h\ m = \mathbb{S}_T\ \$\ \lambda s_2 \rightarrow \mathbf{do}\ (a, s_1) \leftarrow run\mathbb{S}_T\ m\ (h\ s_2)$$
$$return\ (a, g\ s_1)$$

without committing to bi-directional views in particular. In this case *stateIso* converts between $\mathbb{S}_T\ s_1\ m$ and $\mathbb{S}_T\ s_2\ m$ where $s_1$ and $s_2$ are isomorphic. This is useful, for instance, to share a state between components that expect values in different units, or to employ different character or data encodings.

***Isomorphic Read-Only Views*** Consider how we can turn the uni-directional read-only view $r_1$ into a bi-directional one. For that purpose, the target type $\mathbb{R}_T\ s\ m$ is unsuitable because it cannot keep track of updates. So we need to alter the target type for bi-directional read-only views. Fortunately, components are typically polymorphic in the monad stack and do not particularly care about a $\mathbb{R}_T$ view; any instance of $\mathbb{R}_M$ will do. We may put this freedom to good use by defining our own instance that is (trivially) isomorphic with any state monad.

$$\mathbf{newtype}\ \mathbb{S}_M\mathbb{R}_T\ s\ m\ a = \mathbb{S}_M\mathbb{R}_T\ \{run\mathbb{S}_M\mathbb{R}_T :: m\ a\}$$
$$\mathbf{instance}\ \mathbb{S}_M\ s\ m \Rightarrow \mathbb{R}_M\ s\ (\mathbb{S}_M\mathbb{R}_T\ s\ m)\ \mathbf{where}$$
$$ask = \mathbb{S}_M\mathbb{R}_T\ get$$
$$\mathbf{instance}\ MonadTrans\ (\mathbb{S}_M\mathbb{R}_T\ s)$$
$$r :: (\mathbb{S}_M\ s\ m, View\ (\rightsquigarrow)) \Rightarrow \mathbb{S}_M\mathbb{R}_T\ s\ m \rightsquigarrow m$$
$$r = view\ run\mathbb{S}_M\mathbb{R}_T\ \mathbb{S}_M\mathbb{R}_T$$

## 5. Masks

The operations on views presented in Section 4 can be seen as the foundation for a masking language for monad transformers. Using this masking language it is possible to apply a mask to a particular monad stack to hide, restrict access or grant full access to the various layers in the monad stack.

### 5.1 Structural Masking

Consider again the example program of Section 2.4 where two components, *ifpos* and *comp* ≻ *get*, access disjoint layers in the monad stack $M$.

$$\mathbf{type}\ M = \mathbb{S}_T\ Int\ (\mathbb{S}_T\ Int\ (\mathbb{E}_T\ String\ (\mathbb{E}_T\ String\ \mathbb{I})))$$

The former component accesses the second and third layer, while the latter component accesses the first and fourth layer. The snag is that the same state monad transformer type is used for the first and second layer, and the same error monad transformer for the third and fourth layer.

The repeated transformer types suggest to use the monad zipper, but there is a complication. So far we have used the monad zipper to create a *single focal point*, ignoring a prefix of the monad stack. What we need now are *multiple focal points*, ignoring arbitrary parts of the monad stack inbetween. We achieve this by composing multiple zippers into a single view. We call such a view a *structural mask*, a *mask* because it hides particular layers of the monad stack, and *structural* because the form of the mask follows the structure of the monad stack (we will see non-structural masks later).

To facilitate writing structural masks, we formulate them in terms of two primitive (1-layer or 1-bit) masks □ and ■, and one combinator (:::) that associates to the left.

The symbols □ and ■ denote respectively a transparent and an opaque mask, similar to role the bits 1 and 0 play in a bit mask. In our approach, □ means as much as "I want to see the current layer of the monad stack" and ■ means "I don't want to see the current layer". The (:::) combinator, that associates to the left, adds a 1-bit mask at the front of an n-bit mask, similar to the list cons operator (:).

The views framework of Section 4 provides the appropriate infrastructure to implement the mask primitives: The □ mask is nothing more than the identity isomorphism.

$$□ :: (Monad\ m, View\ (\rightsquigarrow)) \Rightarrow m \rightsquigarrow m$$
$$□ = id_M$$

The ■ mask captures another familiar isomorphism, that of the monad zipper.

$$■ :: (MonadTrans\ t_1, MonadTrans\ t_2, Monad\ m, View\ (\rightsquigarrow))$$
$$\Rightarrow (t_1 \triangleright t_2)\ m \rightsquigarrow t_1\ (t_2\ m)$$
$$■ = view \uparrow \downarrow$$

Finally, the composition operator is:

$$(:::) :: (Monad\ m, Monad\ n$$
$$, MonadTrans\ t_1, MonadTrans\ t_2, View\ (\rightsquigarrow))$$
$$\Rightarrow (t_1\ n \rightsquigarrow t_2\ n) \rightarrow (m \rightsquigarrow n) \rightarrow (t_1\ m \rightsquigarrow t_2\ n)$$
$$v_1 ::: v_2 = v_1 \bullet hmap\ v_2$$

So a mask that hides the second and third layer is

$m_2 :: ... \Rightarrow t_1\ (((t_2 \triangleright t_3) \triangleright t_4)\ m) \rightsquigarrow t_1\ (t_2\ (t_3\ (t_4\ m)))$
$m_2 = \square ::: \blacksquare ::: \blacksquare ::: \square$

Similarly, we could write a mask that hides the first and fourth layer as $\blacksquare ::: \square ::: \square ::: \blacksquare$. However, the much shorter

$m_1 :: ... \Rightarrow (t_1 \triangleright t_2)\ m \rightsquigarrow t_1\ (t_2\ m)$
$m_1 = \blacksquare$

has exactly the same effect in this example. In conclusion, the desired composition is

$client :: M\ Int$
$client = from\ m_1\ (ifpos\ (to\ m_1\ (from\ m_2\ (comp \gg get))))$

### 5.2 Nominal Masking

While the structural masking approach above resolves the issue of focusing on different layers, it can be awkward to use and maintain. Masking the stack requires global structural knowledge of the monad stack that is fragile with respect to changes. When the stack layout changes, all masks have to be adjusted accordingly.

As a remedy, we propose a *nominal* masking technique. A nominal mask specifies the names rather than the positions of the monad stack layers that a component may access. The layers in the monad stack are correspondingly tagged with these names. Using the names and the tagged stack structure, the appropriate structural masks are automatically derived. This makes the approach much more robust: when the stack layers are reorganized (e.g. to insert a new layer or to swap two layers), the structural masks are adjusted accordingly.

***Tagged Monad Stack*** The $\mathbb{T}_T$ *tag* monad transformer labels a particular position in the monad stack with a type-level name *tag*.

**newtype** $\mathbb{T}_T\ tag\ m\ a = \mathbb{T}_T\ \{run\mathbb{T}_T :: m\ a\}$

**instance** $MonadTrans\ (\mathbb{T}_T\ tag)$ **where**
  $lift = \mathbb{T}_T$

which can be combined with other monad transformers, e.g. $\mathbb{S}_T$ to create tagged transformers.

**type** $\mathbb{TS}_T\ tag\ s\ m = \mathbb{T}_T\ tag\ (\mathbb{S}_T\ s\ m)$

$run\mathbb{TS}_T :: Monad\ m \Rightarrow tag \rightarrow s \rightarrow \mathbb{TS}_T\ tag\ s\ m\ a \rightarrow m\ (a, s)$
$run\mathbb{TS}_T\ t\ s\ m = run\mathbb{S}_T\ (run\mathbb{T}_T\ m)\ s$

Singleton types are used for tag names, e.g.

**data** $Counter_1 = Counter_1$
**data** $Counter_2 = Counter_2$

The type class $TWith\ tag\ n\ m$ relates a monad stack $n$ with a particular mask $m$ that puts the focus on the layer with name *tag*.

**class** $(Monad\ m, Monad\ n) \Rightarrow TWith\ tag\ n\ m$ **where**
  $structure :: View\ (\rightsquigarrow) \Rightarrow tag \rightarrow (n \rightsquigarrow m)$

We refer to Appendix A for the instances implementing *TWith*; these automatically derive the appropriate structural mask.

Two convenient additional combinators are

$use \quad :: TWith\ tag\ n\ m \Rightarrow n\ a \rightarrow tag \rightarrow m\ a$
$c\ `use`\ name \quad = from_{\bowtie}\ (structure\ name)\ c$
$expose :: TWith\ tag\ n\ m \Rightarrow m\ a \rightarrow tag \rightarrow n\ a$
$c\ `expose`\ name = to_{\bowtie}\ (structure\ name)\ c$

Hence, we may write

$c =$ **do** $inc \quad `use`\ Counter_1$
$\quad (inc \gg inc)\ `use`\ Counter_2$
$\quad return\ ()$

to configure a number of incrementers. The stack layout is easily modified without requiring changes to the component configuration.

$> run\mathbb{I}\ \$\ run\mathbb{TS}_T\ Counter_2\ 5\ \$\ run\mathbb{TS}_T\ Counter_1\ 0\ \$\ c$
$(((), 1), 7)$
$> run\mathbb{I}\ \$\ run\mathbb{TS}_T\ Counter_1\ 0\ \$\ run\mathbb{TS}_T\ Counter_2\ 5\ \$\ c$
$(((), 7), 1)$

The above approach can be extended from masking with a single tag to masking with a type-level list of tags. For instance, $Counter_1\ \&\ Error_1$ represents a mask that views both layers $Counter_1$ and $Error_1$. Here, (&) is the constructor for non-empty type-level lists. We refer to the source code for all the necessary definitions.

Moreover, it nicely integrates with the other views. For instance, $stateIso\ f\ f^{-1} \bullet structure\ Counter$ selects the layer named *Counter* and applies a state isomorphism to it.

### 5.3 Formal Mask Parameters

So far, we have applied masks, and views in general, on components in a non-invasive fashion, from the *outside*. However, views as formal parameters within components also have an important use in disambiguating different instances of an effect. Obviously *externally* applied views are no solution to this problem that already manifests itself inside a component. The traditional solution to disambiguate two different states within a component is to use *lift*.

Consider again the *doubleInc* example in Section 2. To disambiguate the two states, we have used *lift*. Unfortunately, such internally motivated uses of *lift* impose unnecessary ordering constraints: in the monad stack one state transformer must appear above the other. Reversing the order of the two transformers is not possible; for that purpose we need to change the component implementation or write an alternate version.

Explicit view parameters allow us to abstract from the ordering, similarly to *tagged transformers* [19, 25] but with two advantages: *(i)* we get the full expressivity of views for adapting the concrete monad stack, and *(ii)* we get the full expressivity of bi-directional views for handling mutual embedding of components.

$doubleInc_2 :: (MonadCategory\ (\rightsquigarrow), \mathbb{S}_M\ Int\ n_1$
$\qquad , \mathbb{S}_M\ Int\ n_2, Monad\ m)$
$\qquad \Rightarrow (n_1 \rightsquigarrow m) \rightarrow (n_2 \rightsquigarrow m) \rightarrow m\ Int$
$doubleInc_2\ v_1\ v_2 =$ **do** $from\ v_1\ inc$
$\qquad\qquad\qquad from\ v_2\ inc$

So $doubleInc_2\ \square\ \blacksquare$, $doubleInc_2\ \blacksquare\ \square$ and $doubleInc_2\ \square\ \square$ express both orderings of two disjoint states as well as a single shared state.

***Call-by-reference Operations*** It is possible to create variants of effectful operations that take a view argument. For example, the $get_v$ and $put_v$ operations used in the *add* example in Section 2 are defined as:

$get_v\ v = from\ v\ \$\ get$
$put_v\ v = from\ v \circ put$

## 6. Case Study: Monadic Mixins

This section illustrates the uses of the monad zipper and monad views on monadic mixins. Because monadic mixins are higher-order components with non-trivial control flow patterns, traditional mechanisms to manipulate the monad stack do not provide adequate support. However, with the monad zipper and monad views, the complex control-flow patterns of monadic mixins do not pose a problem.

### 6.1 Monadic Mixin Components

***Mixins*** We briefly summarize the notion of mixins, and refer the interested reader to previous literature on the topic for a more in-depth treatment [4]. A simple form of mixins can be easily implemented in Haskell as follows:

```
type Mixin s = s → s

fix :: Mixin s → s
fix a = a (fix a)

(⊛) :: Mixin s → Mixin s → Mixin s
a1 ⊛ a2 = λproceed → a1 (a2 proceed)
```

The type *Mixin s* is a synonym for a function with type $s \to s$ representing open recursion. The parameter of that function is called a *join point*, that is, the point in the component at which another component is added. The operation ⊛ defines component composition. The function *fix* is a fixpoint combinator used for closing, or sealing, an open and potentially composed component.

***Combining monads with mixins*** When combined with monads, mixins allow us to model a simple form of AOP-like advice [17]. However, the control-flow patterns of programs using mixins are complex. For instance, consider the following memoization component and a monadic fibonacci function.

```
memo :: 𝕊M (Map Int Int) m ⇒ Mixin (Int → m Int)
memo proceed x =
  do m ← get
     if member x m
       then return (m ! x)
       else do y ← proceed x
               m' ← get
               put (insert x y m')
               return y

fib :: Monad m ⇒ Mixin (Int → m Int)
fib proceed n =
  case n of
    0 → return 0
    1 → return 1
    _ → do y ← proceed (n − 1)
           x ← proceed (n − 2)
           return (x + y)
```

We can instantiate different monads, using the corresponding run functions of Figure 1, to recover variations of the fibonacci function. For example, the identity monad recovers the effect-free function while a fast fibonacci function is obtained by adding the memo advice and suitably instantiating the state monad:

```
slowfib :: Int → Int
slowfib = run𝕀 ∘ fix fib

fastfib :: Int → Int
fastfib = evalM empty ∘ fix (memo ⊛ fib)

evalM :: s → 𝕊T s 𝕀 a → a
evalM s m = run𝕀 $ run𝕊T m s ≫= return ∘ fst
```

Another component for profiling is

```
prof :: 𝕊M Int m ⇒ Mixin (a → m b)
prof proceed x = do c ← get
                    put (c + 1)
                    proceed x
```

which allows us to count the number of calls to the fibonacci function

```
proffib = evalM 0 ∘ fix (prof ⊛ fib)
```

**Transformer Conflicts** Of course, we would also like to profile the memoized fibonacci function to get an idea of how much more efficient it is.

```
profmemofib :: Int → 𝕊T Int (𝕊T (Map Int Int) 𝕀) Int
profmemofib = fix (prof ⊛ memo ⊛ fib)
```

Unfortunately, the type checker complains that *Int* and *Map Int Int* are distinct types. The problem is that there are two uses of *get* in our components: one in *prof*; and another in *memo*. Due to automatic lifting, both *get* methods read the state from the same top-level $\mathbb{S}_T$, which happens to contain an *Int* value. This is the right thing to do for *prof*, but wrong for *memo* that expects a value of type *Map Int Int*.

### 6.2 Zipping Mixins

The problem above can be solved using the monad zipper to provide a new composition operator ⊗ for mixins.

```
(⊗) :: Mixin (a → t1 (t2 m) b)
    → Mixin (a → (t1 ▷ t2) m b)
    → Mixin (a → t1 (t2 m) b)
c1 ⊗ c2 = λproceed x → c1 (↑ ∘ c2 (↓ ∘ proceed)) x
```

This combinator associates the left-to-right order of components with a corresponding top-to-bottom order of monad layers. Here component $c_1$ focuses on the current layer, and $c_2$ looks one position down – that's why we have to bring *proceed* down (↓) to its level and shift the whole back up (↑) to the current level.

This combinator is very useful whenever we have a set of components that uses a disjoint set of monads (that is, each component will use different monads). No additional work is needed to make the two state transformers of *prof* and *memo* happily coexist.

```
profmemofib :: Int → 𝕊T Int (𝕊T (Map Int Int) (𝕀T 𝕀)) Int
profmemofib = fix (prof ⊗ memo ⊗ fib)
```

Note that every component has its own transformer, notably $\mathbb{I}_T$ for *fib*, and we use the base monad $\mathbb{I}$ at the bottom of the stack.

### 6.3 Views and Masks

When using mixins we generally need the full power of bidirectional views to shift between two isomorphic monads. To make this shifting convenient we use the following combinator:

```
fmask :: (Monad m, Monad n)
      ⇒ (n ⋈ m) → Mixin (a → n b) → Mixin (a → m b)
fmask v mix proceed = from v ∘ mix (to v ∘ proceed)

mix `uses_m` names = fmask (structure names) mix
```

The *fmask* combinator takes a view *v* and applies it to a mixin *mix* executing the *to* function after proceeding and the *from* function after the mixin. Nominal views are applied with $uses_m$.

To demonstrate the application of masks and views on monadic mixin components, consider a simple assertion component that is used to check the output of a computation. We will show how this component is useful for checking whether the result of computing the fibonacci function has overflowed or not.

```
assertDump :: (𝔼M String m, ℝM s m, Show a, Show b, Show s)
           ⇒ (b → Bool) → Mixin (a → m b)
```

The *assertDump* component applies an assertion (a function of type $b \to Bool$) to the output of *proceed*'s computation. If the assertion fails, an error is raised. The error message includes information on the state at the time of the error, to facilitate debugging.

This *assertDump* component allows the creation of another variant of the fibonacci program where, along with memoization and profiling, we also check for overflow and dump the memo table

when overflow happens. Because we may be interested in recovering from the overflow (for example, by changing the representations of the inputs and outputs from *Int* to *Integer*) and continuing the computation, the exception layer should be at the top of the stack. Also, profiling works better if it is executed before memoization, so the profiler component should be in between *assertDump* and *memo*. However, *assertDump* requires access to the monadic layer with the memo table. In order to combine these components together we should make sure that all the constraints are satisfied.

Because there is no simple one-to-one correspondence between mixins and stack layers, we use the nominal approach. In addition, the *r* imposes a read-only view on the *Memo* state for *assertDump*.

```
test :: Int → ((Either String Int, Int), Map Int Int)
test = myEval ∘
  fix ( assertDump (⩾ 0) `fmask` r `uses_m` Err & Memo
      ⊛ prof             `uses_m` Prof
      ⊛ memo             `uses_m` Memo
      ⊛ fib)
myEval m = runI $ runTS_T Memo empty
                $ runTS_T Prof 0
                $ runTE_T Err $ m
```

### 6.4 Monadic Mixins in Practice: Search Combinators

Schrijvers et al. [24] present a compelling application of the monadic mixins. They provide a Domain Specific Language (DSL) for expressing a complex search heuristic as a concise combination of primitive combinators. In their implementation, each primitive combinator corresponds to a monadic mixin component. The advantage and novelty is twofold: 1) *flexibility* because the user is able to combine combinators any way she wants, and 2) *extensibility* because the system developer is able to add new combinators without touching the existing ones.

To fit in with the Gecode C++ Constraint Programming library and for performance reasons, a staged approach is taken. The Haskell mixin components are code generators that collaborate to produce the C++ code for the overall search algorithm. For example, a search heuristic tailored to solving radiation therapy planning problems [1] of typical size consists of 20-30 monadic mixins, each with their own monad transformer, that collaborate to produce around 2000 lines of specialized C++ code. We are not aware of other Haskell projects with comparable monad stack sizes.

***Effect Encapsulation*** Additionally to the techniques already introduced by us earlier in this section, the search combinators application employs a technique that facilitates the dynamic composition of many mixin components from a parsed specification string: each component encapsulates its own effect. Even statically this encapsulation makes sense. After all, the type of a monad stack with 20 layers is rather unwieldy.

The *Component* datatype below illustrates the encapsulation technique on the simple *Mixin* type.[6] The effect $t_2$ of the component is existentially quantified to hide it, and the included *run* function allows eliminating it.

```
data Component a b = ∀ t2.MonadTrans t2 ⇒
  Component {behavior :: ∀ t1 m.(MonadTrans t1, Monad m)
                      ⇒ Mixin (a → (t1 ▷ t2) m b)
            , run     :: ∀ m x.Monad m ⇒ t2 m x → m x}
```

While not all monad transformers support a *run* function of the above type, it is convenient and sufficient for our search combinators. For more details on the encapsulation technique and a more general form of *run* function, we refer to [22, Section 3.4].

[6] The mixin record type of search components is too elaborate to show here.

***Performance Considerations*** We have not performed any systematic benchmarks yet, but do have a few observations on performance. Because it is defined as a `newtype`, wrapping and unwrapping the monad zipper does not add any space or runtime overhead. It does generate different type class dictionaries for the different components, and depending on the amount of inlining this either happens once (statically) or repeatedly for each invocation of a component (dynamically). The latter scenario notably arises when the effect types are existentially quantified. We have observed that this repeated creation of dictionaries puts compositions with around 60 monad transformers out of reach. In conclusion, gracefully and predictably scaling performance to monad stack sizes well beyond 30 layers is an open challenge.

## 7. Discussion and Related Work

### 7.1 The Haskell Setting

Our approach makes use of two key ingredients, monadic types and constrained polymorphism, which are both available in Haskell. The combination of these provides the necessary flexibility for effectful components to be adapted to many different settings.

One important advantage of using of Haskell is type-inference. In a setting like ours, where types can be relatively complex, type-inference is a blessing and allows most types to be inferred automatically. Indeed, although we have often used type annotations in our examples for documentation, those annotations are (for the most part) not necessary.

While Haskell provides the two necessary ingredients, there is potential for transferring the presented ideas to other settings. The monad zipper, views and structural masks have category theory interpretations and as such are of a more general nature. Indeed, they are relevant and adaptable to other settings that deal with explicit effects. Haskell-specific implementation aspects can find alternative counterparts in other languages. For instance, in Scala *implicits* can replace type classes [16]. The type-class tricks to look up names for nominal masks, popularized by Kiselyov [12], could be considerably simplified in a dependently typed language. Finally, a language design that natively supports the presented concepts is another option.

### 7.2 Effect Systems and Modular Monads

Effect systems (also known as type-and-effect systems) [14] form a popular non-monadic approach for making side effects explicit. However, they only describe (and do not define) programs that already have a meaning independent of the effect system. Hence, the effect annotations cannot adapt component behavior. Filinski's MultiMonadic MetaLanguage ($M^3L$) [7, 8] does embrace the monadic approach, but uses subtyping (or subeffecting) to combine the effects of different components. The subtyping relation is fixed at the program or language level, which does not provide the adaptability we achieve with constrained polymorphism.

Since Moggi [15] proposed monads to model side-effects, and Wadler [30] popularized them in the context of Haskell, various researchers (e.g., [11, 26]) have sought to modularize monads. Monad transformers emerged [3, 13] from this process, and in later years various alternative implementation designs, facilitating monad (transformer) implementations, have been proposed, such as Filinksi's layered monads [6] and Jaskelioff's Monatron [10].

In this paper, we rely on several of Monatron's techniques for implementing monad transformers. However, so as not to confuse the reader, we have incorporated the necessary techniques form Monatron in the familiar setting of the Monad Transformer Library[7] (MTL).

[7] which implements the original ideas of [13]

In our variant of the MTL, [8] monad transformers have to supply two additional operations, *tmap* and *mw*, which are inspired by Monatron:

```
class MonadTrans t where
  lift  :: Monad m ⇒ m a → t m a
  tmap  :: (Monad m, Monad n)
        ⇒ (∀x.m x → n x) → t m a → t n a
  mw    :: Monad m ⇒ MonadWitness t m
```

Here, *tmap* allows transforming the monad underneath a transformer *t*. We have two uses for this additional operation:

- Our first use, just like in Monatron, is to lift control operators like *catchError* and *local* through other transformers; in the original MTL these operations could not be lifted. In our setting, it enables us to implement these control operators appropriately for the zipper ($\triangleright$).

  For instance, here is the zipper implementation of the *local* operator from the $\mathbb{R}_{\mathrm{M}}$ type class:

  $$local\ f\ m = \mathbb{Z}_{\mathrm{T}}\ \$\ tmap\ (local\ f)\ \$\ run\mathbb{Z}_{\mathrm{T}}\ m$$

- The second use, already covered in Section 4, is for the implementation of the *hmap* function of the *MonadCategory* type class.

The *mw* method encodes the property $\forall m.Monad\ m \Rightarrow Monad\ (t\ m)$ as a GADT witness.

```
data MonadWitness t m where
  MW :: Monad (t m) ⇒ MonadWitness t m
```

In the original MTL, this property holds informally, but is neither enforced nor exploitable. We require this property to implement many of the zipper's operations. For instance, the *return* method for the zipper is defined as:

```
return x = case (mw :: MonadWitness t2 m) of
  MW → case (mw :: MonadWitness t1 (t2 m)) of
    MW → ℤT $ return x
```

In the Monatron design, the property is directly conveyed to the type checker in the form of a single type class instance:

```
instance (Monad m, MonadTrans t) ⇒ Monad (t m) where ...
```

However, this approach is not compatible with the MTL design. Hence, our GADT witness approach.

A limitation of *tmap*, both in Monatron and our MTL variant, is that it only works for covariant monad transformers. For instance, the contravariant continuation monad transformer cannot implement this operation.

### 7.3 Tag-indexed monads

It has been independently suggested by Snyder & Alexander [25] and Piponi [19] that indexing monad transformers with a type-level tag improves their convenience and robustness to changes. MTLX [25] is a monad transformer library that embodies this idea: automatic lifting ambiguities are resolved by tags. For example, consider the programs $p_1$ and $p_2$ written, respectively, in the MTL and MTLX.

```
p1 :: (𝕊M Int m, 𝕊M Bool m) ⇒ m Int
p1 = do b ← get
        x ← get
        return (if b then x else 0)
```

[8] available at `http://users.ugent.be/~tschrijv/Haskell/MTLzipper.tgz`.

```
p2 :: (𝕊M Ix1 Int m, 𝕊M Ix2 Bool m) ⇒ m Int
p2 = do b ← get Ix1
        x ← get Ix2
        return (if b then x else 0)
```

The $p_1$ program does not type-check because the functional dependencies in the type class $\mathbb{S}_{\mathrm{M}}$ require that the state type is uniquely determined by *m*, but in this program two different state types are used. The program $p_2$, written for MTLX, solves this issue by using a $\mathbb{S}_{\mathrm{M}}$ class with a third type parameter, which is used to index the monad and determine the state type (along with the monad type), which avoids the conflict in $p_1$.

The main advantage of indexed monads is that they simplify the implementation of monadic components with multiple instances of the same effect. However tags commit to global names, which restricts reuse. If the lack of flexibility is not an objection, tags are a good solution. However, when the primary focus is flexibility and reusability, tags have important drawbacks compared to the monad zipper and monad views. In the MTLX approach, it is possible to abstract over the tag in order to be able to choose the monad layer later. For example:

```
inc :: (𝕊M ix Int m) ⇒ ix → m Int
inc ix = do x ← get ix
            put ix (x + 1)
            return (x + 1)
```

This value-level abstraction is more verbose than the monad zipper's type-level abstraction; it is an invasive approach that requires the component to be written with tags in mind. Moreover, for the price of value-level abstraction, we get a lot more expressivity from views: restricting access to layers is not possible with tags. Finally, a predefined set of tags is unsuitable when components are generated and composed dynamically, as in our search combinators application.

Ultimately, indexed monads and our techniques are useful for solving different problems and it is possible to get the benefits of both approaches by combining some of the techniques. For example, if we added monad views and the monad zipper to MTLX, we could use views to virtualize the tag names, by renaming or even removing the tags to suit the client's monad stack, and effectively acting as a scoping mechanism for the name tags.

### 7.4 Monadic Components

Many works have identified a need for reusable monadic components, but have not addressed the limitations related to monad stack management. Mixins were introduced by Cook [4] as a functional form of inheritance. Brown and Cook [2] first considered monadic mixins for memoization, while Oliveira et al. [17] model arbitrary AOP-style advice with them and show how to reason about interference between two components based on equational reasoning and parametricity. Our work should enable a generalization of the latter reasoning results to scenarios with arbitrarily many components. Prehofer [20] also considers a monadic model for FOP, which is not based on mixins.

The techniques presented in this paper can be used to improve the current state-of-the-art approaches to modular interpreters [5, 13, 27]. We describe a modular effectful interpreter case study, which uses the techniques presented in this paper, in a separate manuscript [23].

## 8. Conclusion

The current-state-of-the-art in monad stack management is too restrictive to effectively support reusable monadic components. The monad zipper provides the basic mechanism to lift these restrictions, and enables more powerful solutions such as structural and

nominal masking. Views unite masks and other monad transformations in a single framework for adapting monadic components to a wide range of monad stacks.

## Acknowledgments

We are grateful to the anonymous reviewers, Jeremy Gibbons, heisenbug, Mauro Jaskelioff, Wonchan Lee, Wouter Swierstra, Tarmo Uustalu, Phil Wadler, Stephanie Weirich and the members of IFIP WG 2.1 for their help and feedback.

Bruno Oliveira is supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant R11-2008-007-01002-0 and the Mid-career Researcher Program (2010-0022061) through NRF grant funded by the MEST.

## References

[1] D. Baatar, N. Boland, S. Brand, and P. Stuckey. CP and IP approaches to cancer radiotherapy delivery optimization. *Constraints*, 2011.

[2] D. Brown and W. R. Cook. Monadic memoization mixins. Technical Report TR-07-11, The University of Texas, 2007.

[3] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *CCTCS '93: Proceedings of the Conference on Category Theory and Computer Science*, 1993.

[4] W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[5] L. Duponcheel. Writing modular interpreters using catamorphisms, subtypes and monad transformers. Technical report, Utrecht University, 1995.

[6] A. Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 175–188, 1999.

[7] A. Filinski. On the relations between monadic semantics. *Theor. Comput. Sci.*, 375:41–75, April 2007.

[8] A. Filinski. Monads in action. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 483–494, 2010.

[9] G. Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[10] M. Jaskelioff. Monatron: An extensible monad transformer library. In *IFL '08: Symposium on Implementation and Application of Functional Languages*, 2008.

[11] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New Haven, Connecticut, USA, December 1993.

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell 04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107, 2004.

[13] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'95*, 1995.

[14] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, 1988.

[15] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[16] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA '10:Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 341–360, 2010.

[17] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: Disciplined advice with explicit effects. In *AOSD '10: ACM SIG Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 2010.

[18] S. Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[19] D. Piponi. Tagging monad transformer layers, 2010. `http://blog.sigfpe.com/2010/02/tagging-monad-transformer-layers.html`.

[20] C. Prehofer. Flexible construction of software components: A feature oriented approach. Habilitation Thesis, Fakultät für Informatik der Technischen Universität München, 1999.

[21] J. C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, 83:513–523, 1983.

[22] T. Schrijvers and B. C. d. S. Oliveira. Modular components with monadic effects. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010)*, number UU-CS-2010-020, pages 264–277, 2010.

[23] T. Schrijvers and B. C. d. S. Oliveira. The monad zipper. Report CW 595, Dept. of Computer Science, K.U.Leuven, 2010.

[24] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. Stuckey. Search combinators. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, 2011.

[25] M. Snyder and P. Alexander. Monad factory: Type-indexed monads. In *TFP 2010: Preproceedings of Trends in Functional Programming*, pages 106–120, 2010.

[26] G. L. Steele, Jr. Building interpreters by composing monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 472–492, 1994.

[27] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[28] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87*, pages 307–313, 1987.

[29] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. 1989.

[30] P. Wadler. Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, August 1992.

[31] P. Wadler. The essence of functional programming. In *POPL '92: Principles of Programming Languages*, pages 1–14, 1992.

[32] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

## A. TWith Instances

In this appendix, we provide the implementation of the overloaded *structure* function as a set of five overlapping instances for the *TWith* class. Three instances cover the three possible cases and two instances resolve overlaps between the three possible cases.

Although the implementation below may look daunting to the uninitiated, it relies only on folklore type-level programming techniques. Moreover, the user is never directly confronted with these instance implementations.

```
    -- [1] tag at the top
instance (Monad m, m~n) ⇒ TWith tag n (T_T tag m) where
  structure _ = t

    -- auxiliary clause, to resolve overlap between [1] and [3]
instance (Monad m, m~t n, MonadTrans t)
    ⇒ TWith tag m (T_T tag (t n)) where
      structure _ = t


    -- [2] tag in focus
instance (Monad m, Monad n, MonadTrans t, m~t n)
    ⇒ TWith tag m ((t ▷ T_T tag) n) where
```

```
    structure _ = case (mw :: MonadWitness t (𝕋 tag n)) of
                    MW → ■⁻¹ • hmap t

  -- auxiliary clause, to resolve overlap between [2] and [3]
instance (Monad (t' n), Monad m, Monad n, MonadTrans t,
          m~(((t ▷ 𝕋 tag) ▷ t') n), MonadTrans t')
  ⇒ TWith tag m ((t ▷ 𝕋 tag) (t' n)) where
    structure _ = case (mw :: MonadWitness t' n) of
                    MW → ■

  -- [3] shift focus down
instance (Monad (t0 (t1 n)), Monad m, Monad n,
          TWith tag m ((t0 ▷ t1) n), MonadTrans t0, MonadTrans t1)
  ⇒ TWith tag m (t0 (t1 n)) where
    structure tag =
      case (mw :: MonadWitness t1 n) of
        MW → case (mw :: MonadWitness t0 (t1 n)) of
               MW → ■ • structure tag
```

The above instances make use of the following auxiliary functions:

```
t :: View (⇝) ⇒ m ⇝ 𝕋 tag m
t = view 𝕋 run𝕋
■⁻¹ = view ↓ ↑
```